

Lecture 3

Image Classification

IMAGE PROCESSING AND COMPUTER VISION – PART 2

SAMUELE SALTI

Image Classification

Input



Output

Choose among
these categories

Dog

Cat

Bird

Frog

Person

Some challenges



Intraclass variations



Background clutter

The most difficult even with modern models



Viewpoint variations



Illumination changes

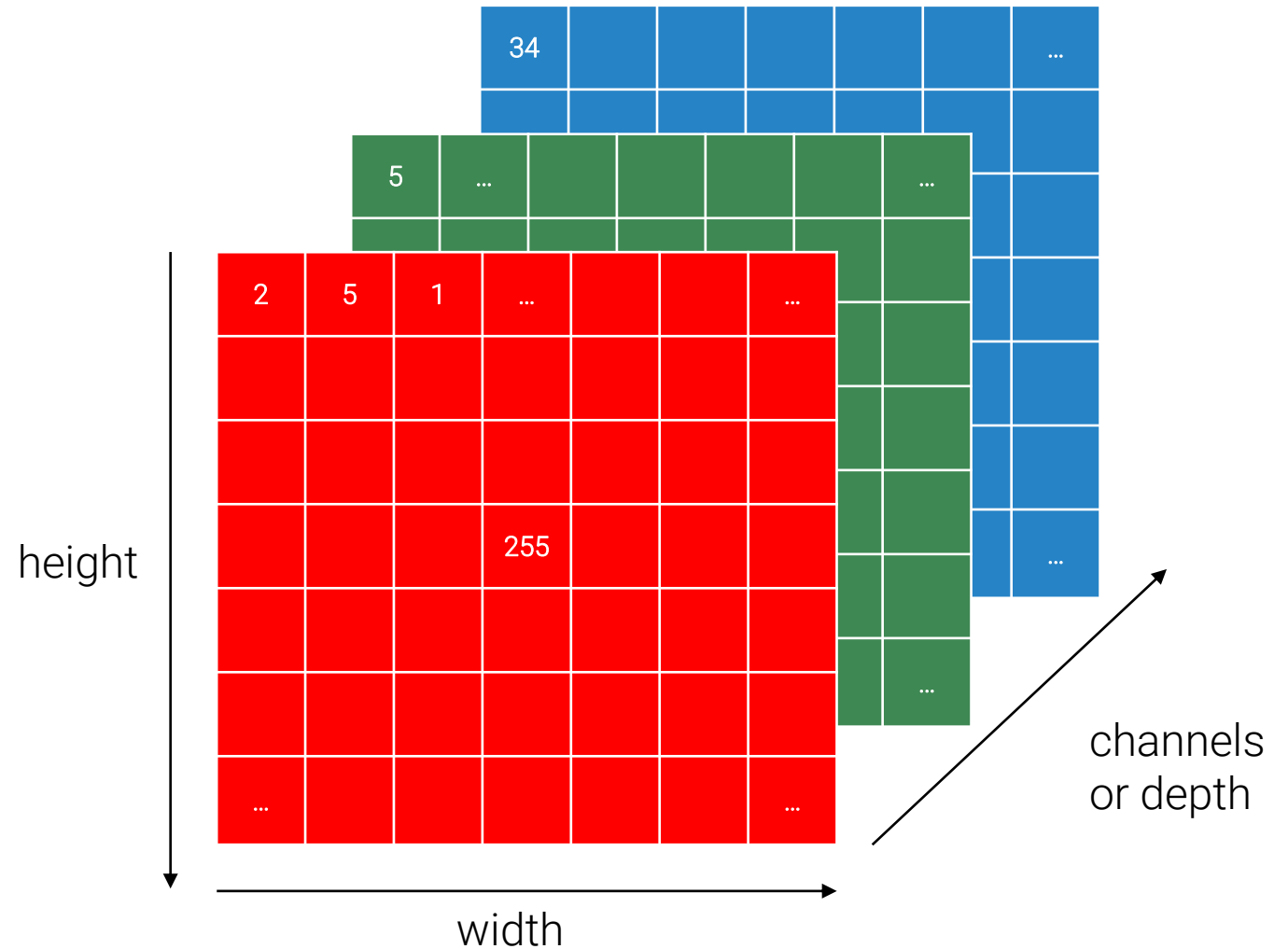


Occlusions



General weirdness of the world...

RGB images are tensors in a computer



Categories as numbers

$$f(\text{img}) = 2$$
A photograph of a white-bellied bluebird (Ptilinopus melanotis) perched on a brown branch. The bird has a bright red beak, white plumage on its head and neck, and vibrant blue feathers on its wings and back. The background is a blurred green, suggesting foliage.

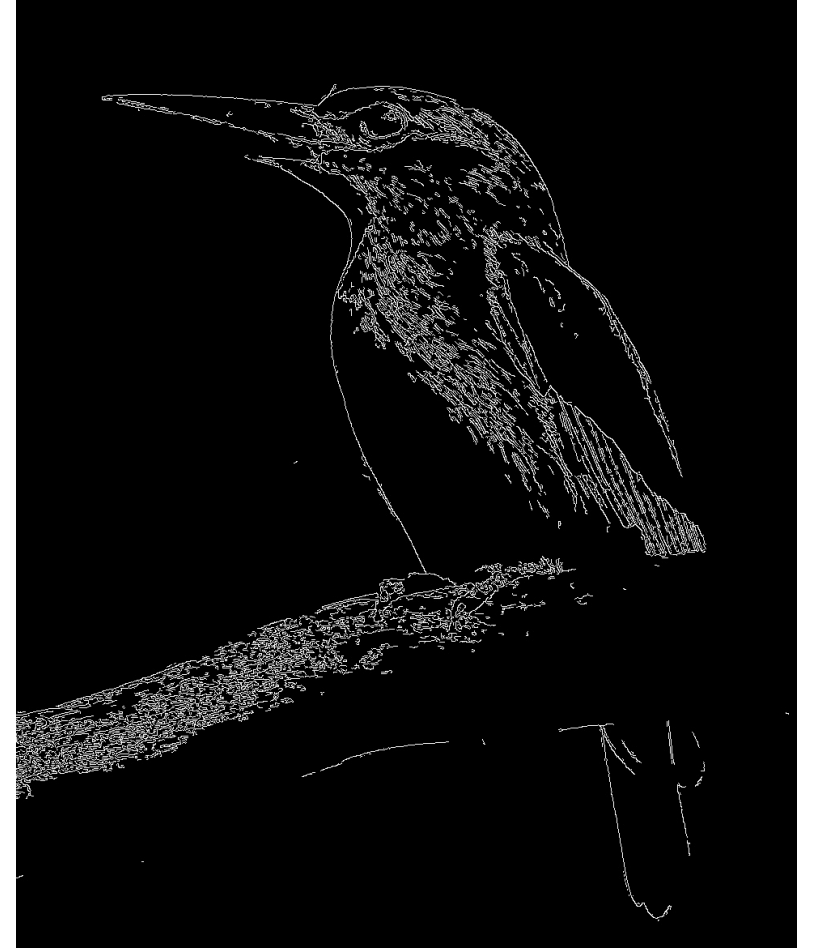
0 -> Dog
1 -> Cat
2 -> Bird
3 -> Frog
4 -> Person

How far can «classic» computer vision bring us?

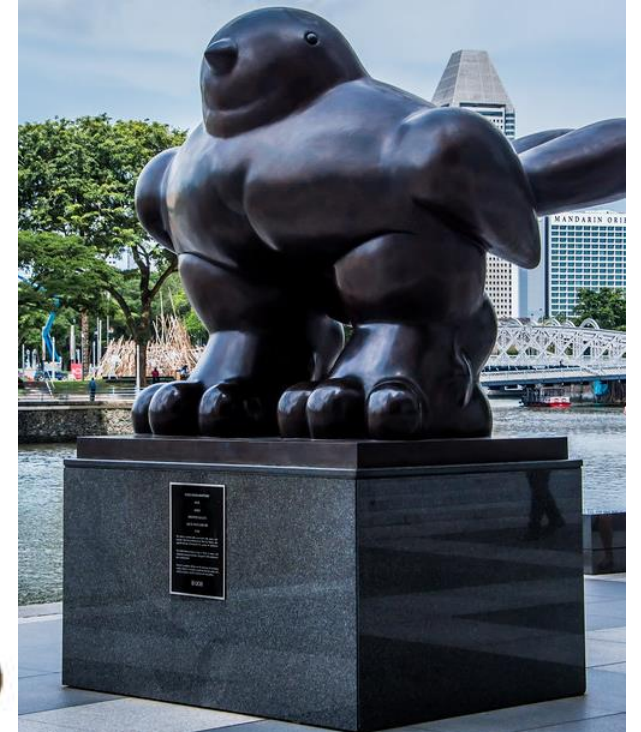
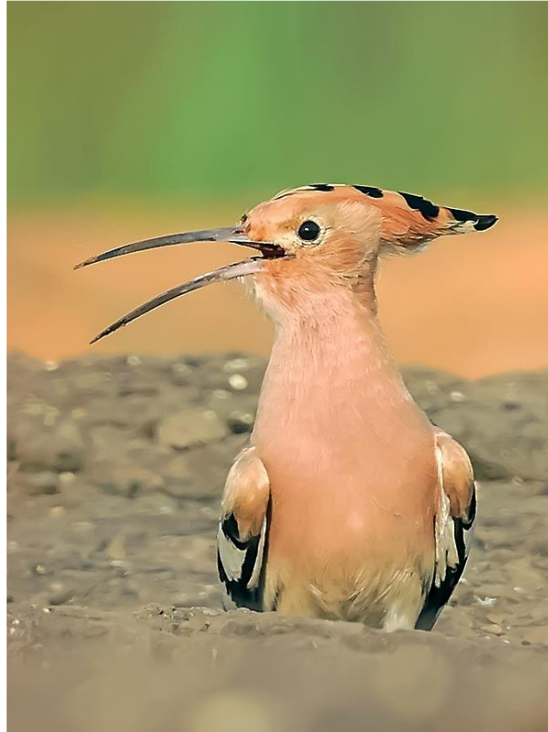


`detect_edges(image)`

If ???



Birds...



Traditional Computer Vision techniques, e.g. handcrafted rules based on edges, need **a controlled environment**, **usually feasible in industrial vision applications**, otherwise they are very brittle.

(Supervised) Machine learning to the rescue

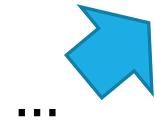
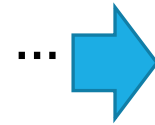
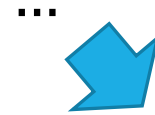
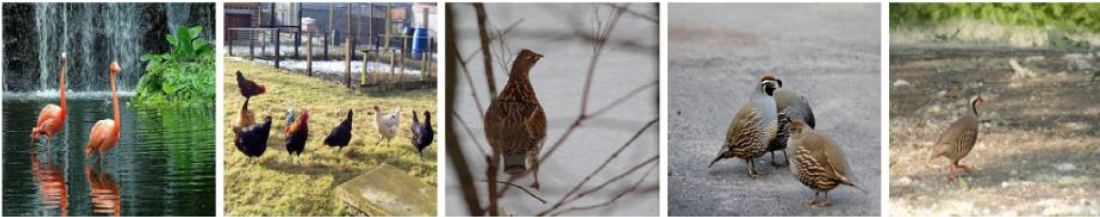
Dog



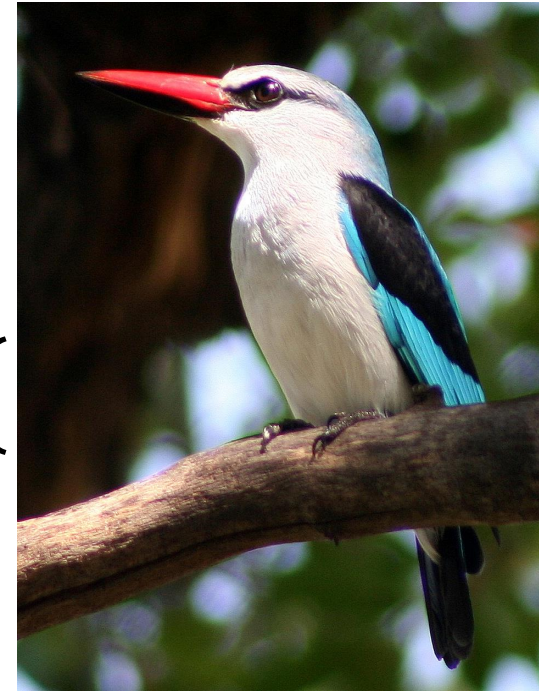
Cat



Bird



$$f(\text{image}) = 2$$



Training and testing dataset

When applying machine learning methods, we are given (or we create):

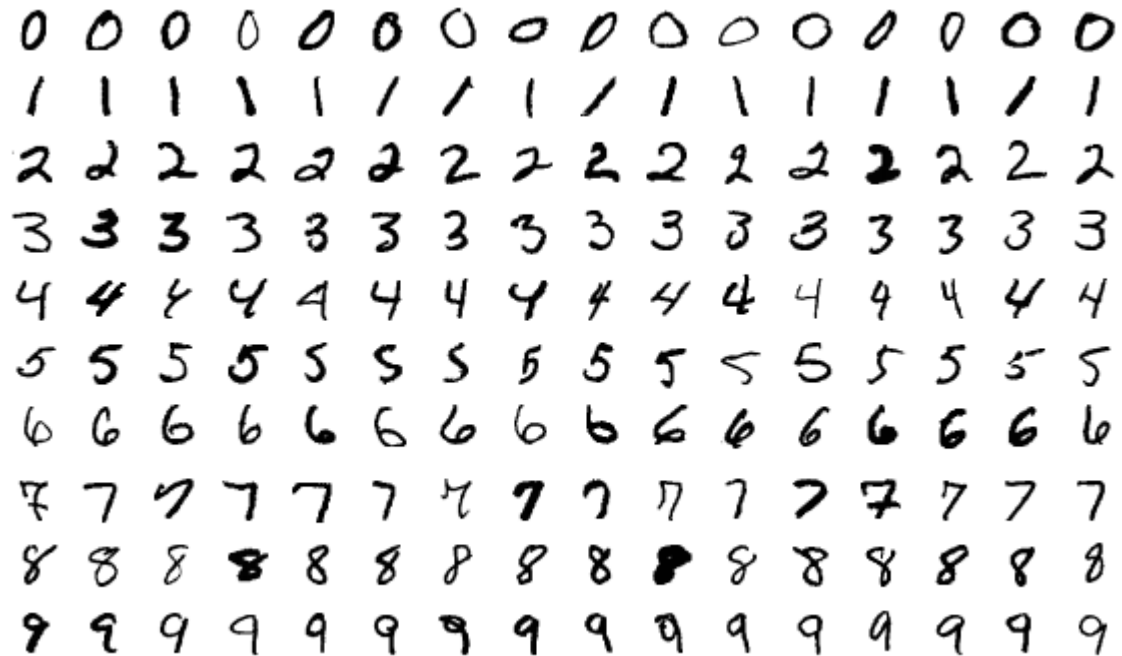
- a training set $D^{train} = \{ (\mathbf{x}^{(i)}, y^{(i)}) \mid i = 1, \dots, N \}$
- a test set $D^{test} = \{ (\mathbf{x}^{(i)}, y^{(i)}) \mid i = 1, \dots, M \}$

where $\mathbf{x}^{(i)} \in \mathbb{R}^f$, are the features representing the real word **items** we care about (i.e. images in our case), and $y^{(i)}$ are the outputs we want to predict for that item, i.e. the **label** in image classification.

We assume the two sets contains **independent and identically distributed** samples from the **same** unknown distribution $p_{data}(\mathbf{x}, y)$

The place in which we extract a dataset about also an object really makes the difference and puts a gap if compared with other nations (ex. bottles in USA vs Argentina)

Modified NIST (MNIST)



10 classes: handwritten digits from 0 to 9

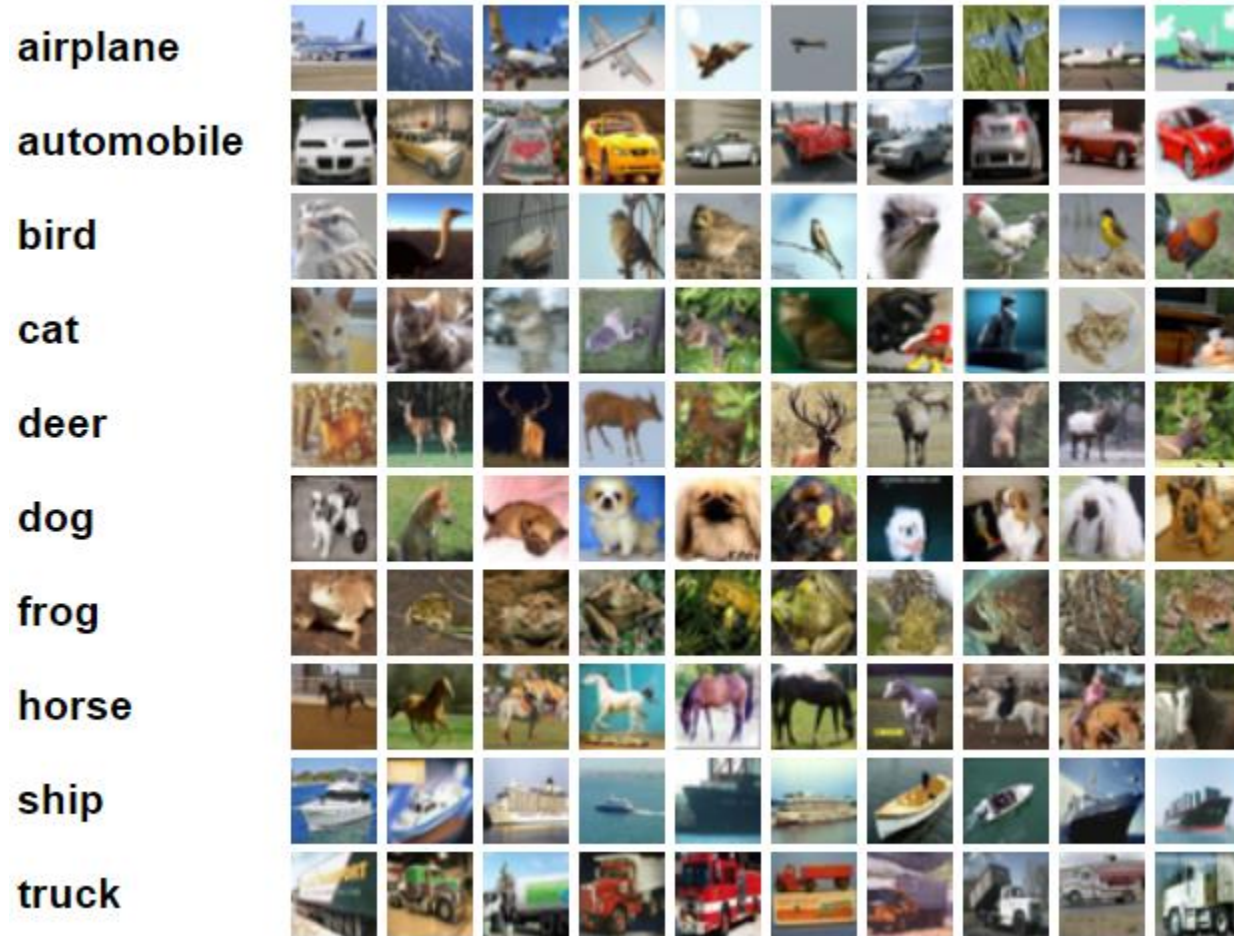
50k train images

10k test images

28x28 grayscale images

Results on MNIST may not generalize to other datasets

CIFAR 10



Subset of the 80 million Tiny Images dataset

<https://www.cs.toronto.edu/~kriz/cifar.html>

10 classes

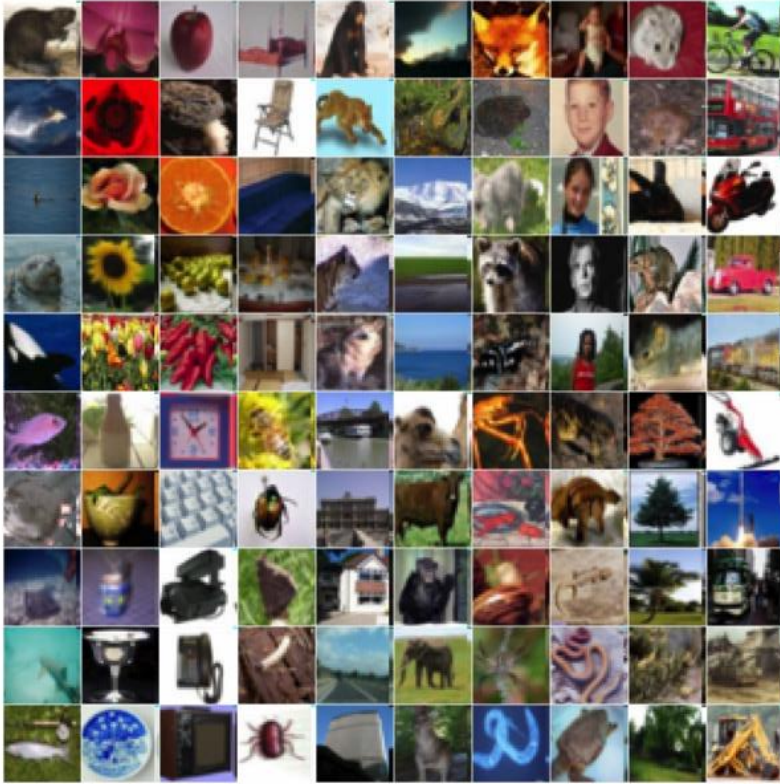
50k training images

10k testing images

32x32 RGB images

[Learning Multiple Layers of Features from Tiny Images](#), Alex Krizhevsky, 2009.

CIFAR 100



Another subset of the 80 million Tiny Images dataset

100 classes

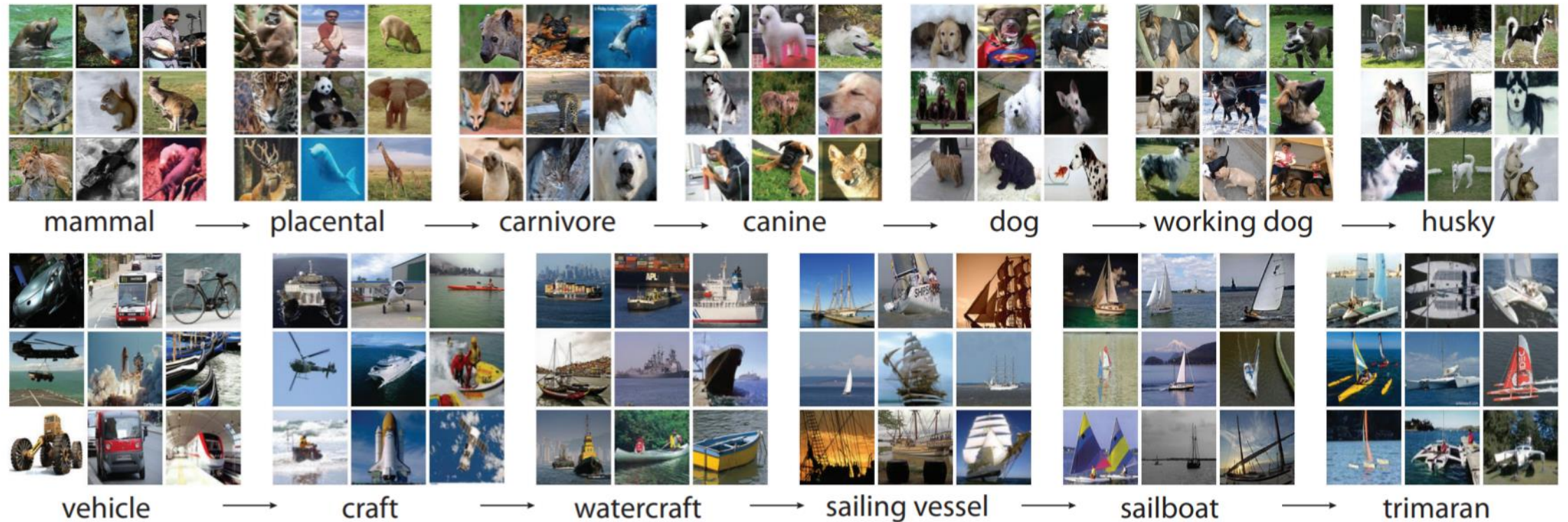
50k training images (500 per class)

10k testing images (100 per class)

32x32 RGB images

Hierarchical structure: 20 super-classes with 5 sub-classes each

ImageNet / ImageNet 21k



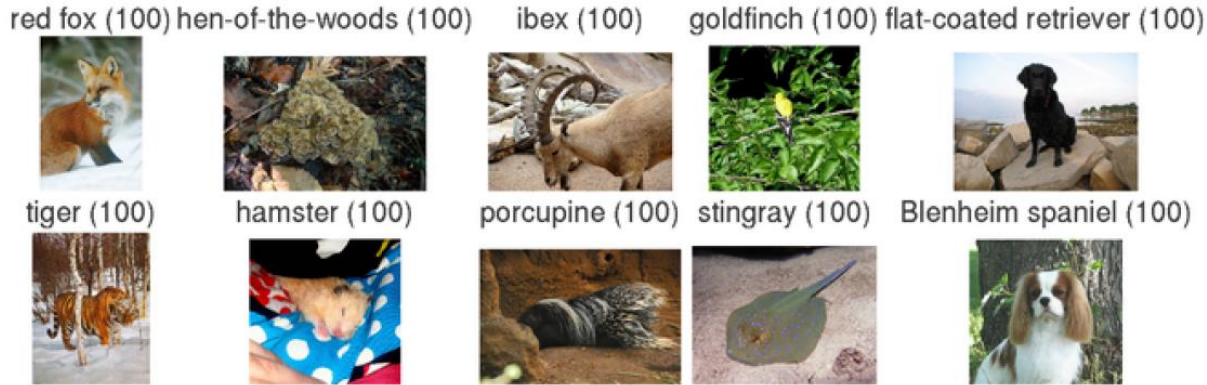
14 millions RGB images at full and variable resolution with average size about 400×350 .

Hierarchical structure: modelled on about 21k synsets from **WordNet** (out of 50k)

ILSVRC – often referred to as ImageNet / ImageNet1k

Image classification

Easiest classes



Hardest classes



1000 classes

1.3M training images (about 1300 per class)

50k validation images (50 per class)

100k test images (100 per class)

Variable resolution RGB images as in ImageNet, often resized to 256x256 for training.

Due to the inherent ambiguity of assigning only one label to each image, performance is usually reported as **top-5 accuracy**: an image is considered correctly classified if the correct label is present in 5 classes predicted by the algorithm. **Multi-label** accuracy has also been recently proposed

<https://cs.stanford.edu/people/karpathy/ilstvrc/>

























Russakovsky et al, "ImageNet Large Scale Visual Recognition Challenge", IJCV 2015
Vaishaal Shankar et al., Evaluating Machine Accuracy on ImageNet, ICML 2020

And many more...

A great resource to find datasets together with state-of-the-art methods is <https://paperswithcode.com/sota>

Benchmarks

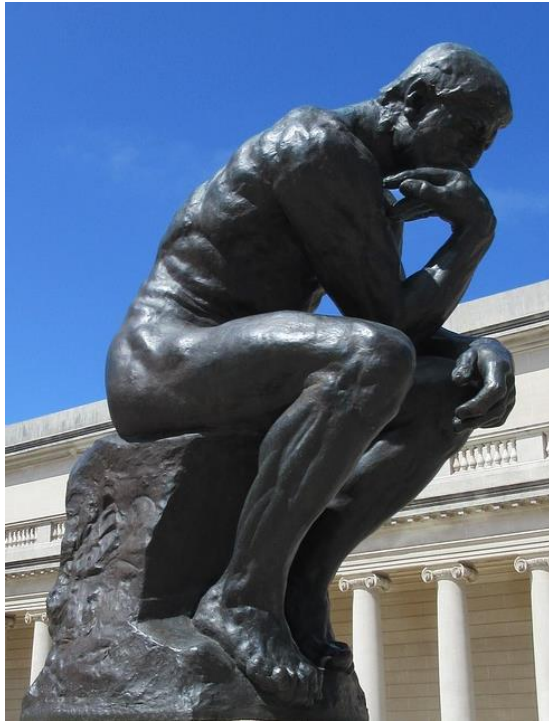
+ Add a Result

TREND	DATASET	BEST METHOD	PAPER TITLE	PAPER	CODE	COMPARE
	ImageNet	 FixEfficientNet-L2	Fixing the train-test resolution discrepancy: FixEfficientNet			See all
	CIFAR-10	 BiT-L (ResNet)	Big Transfer (BiT): General Visual Representation Learning			See all
	CIFAR-100	 BiT-L (ResNet)	Big Transfer (BiT): General Visual Representation Learning			See all
	MNIST	 Branching/Merging CNN + Homogeneous Filter Capsules	A Branching and Merging Convolutional Network with Homogeneous Filter Capsules			See all
	SVHN	 WideResNet-28-10	RandAugment: Practical automated data augmentation with a reduced search space			See all
	STL-10	 NAT-M4	Neural Architecture Transfer			See all

A bird's eye view of traditional programming

Requirements (e.g. sort array)

(May include a few input/output examples, but only for clarity)



Program
`for (int i=0; i < N; i++)`
...

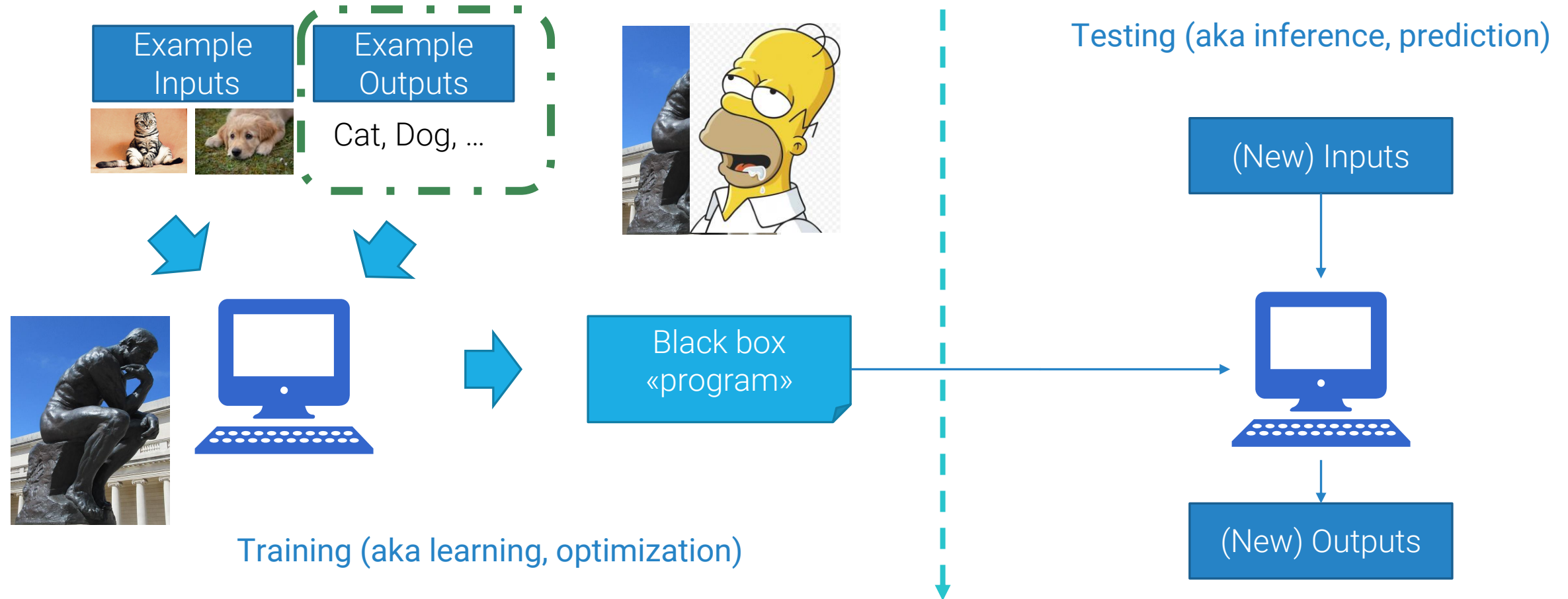
Input
[10,4, 6,1]



Output
[1,4,6,10]

Machine learning or data-driven approach

We can think of machine learning as a new way to instruct computers about what we want them to do.



Data-driven approach: other consequences

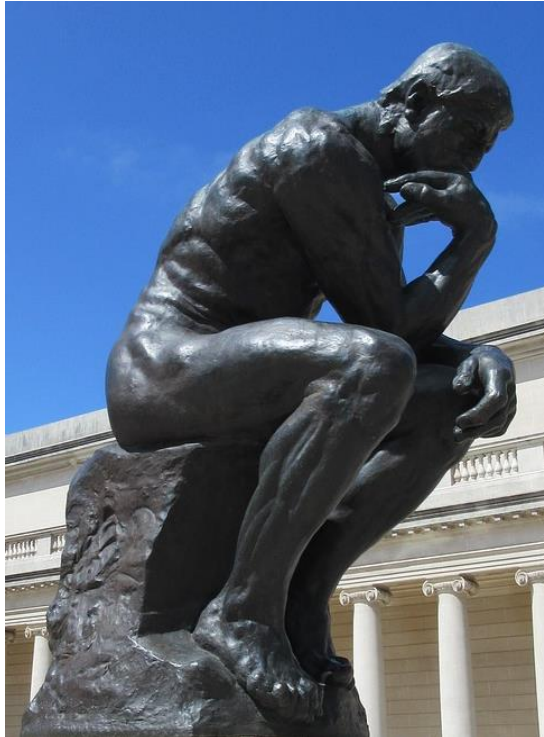
The rise of data-driven approaches changes the relative importance of data and algorithms

Traditional programming

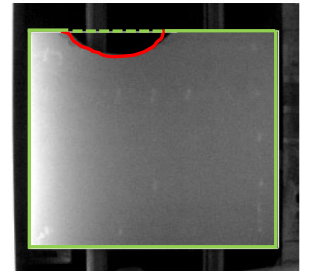


```
for (int i = 0; ...
```

≈



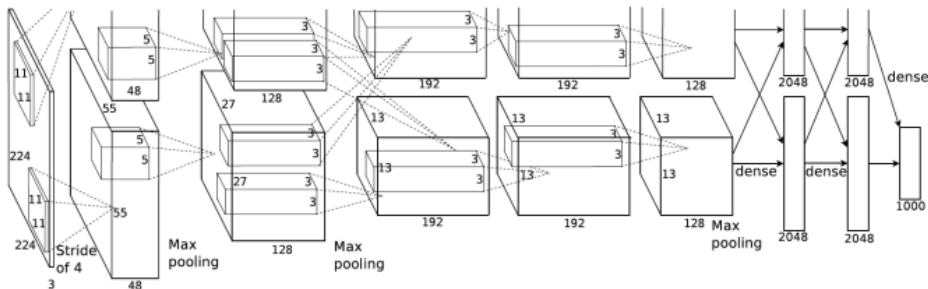
Input
[10,4, 6,1]
Output
[1,4,6,10]



Data-driven approach: other consequences

Data and datasets are crucial in this new paradigm. To know the most used ones for each vision task and to **understand the impact of metrics** on the results is therefore at least as important as it is to know the latest and greatest machine learning models.

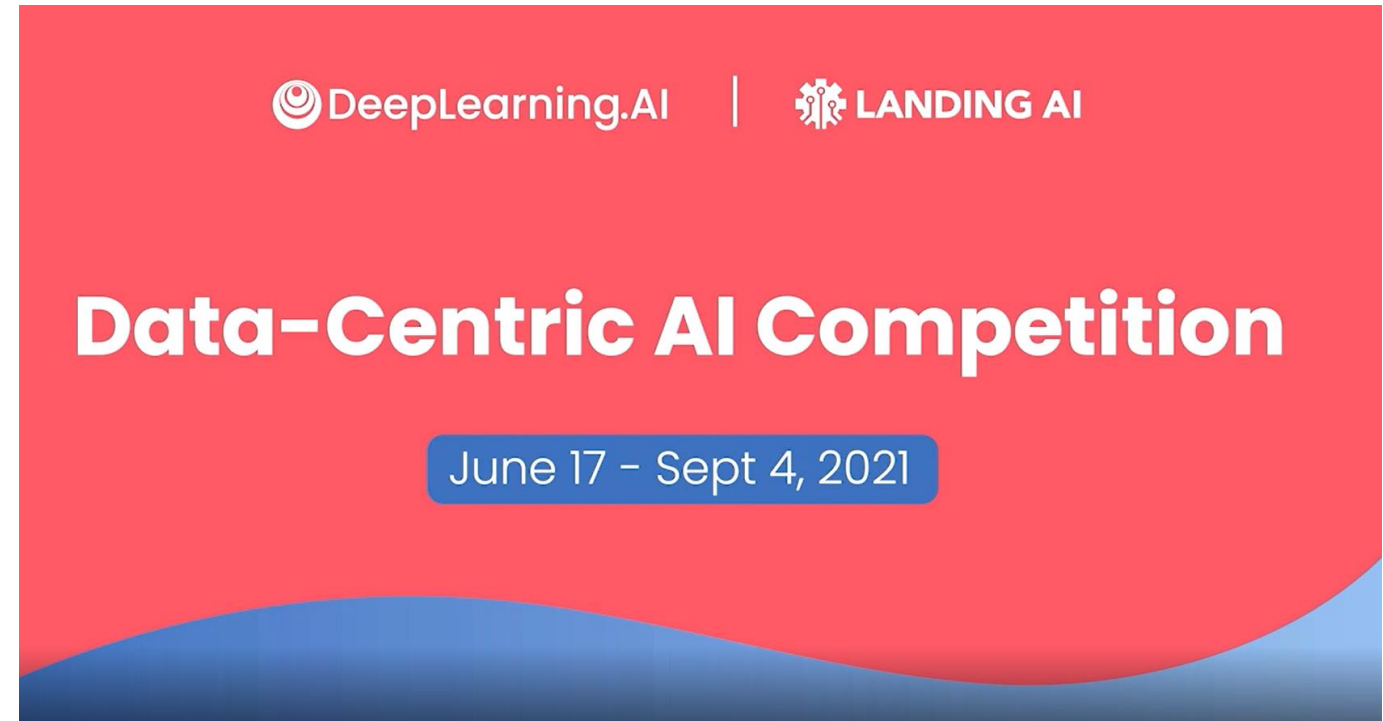
Data-driven approach



MLOps: From Model-centric to Data-centric AI



“If 80% of machine learning is data preparation, we should invest more in it and be more systematic about it”



<https://www.youtube.com/watch?v=06-AZXmwHjo>
<https://worksheets.codalab.org/worksheets/0x7a8721f11e61436e93ac8f76da83f0e6>

Parametric approach

$$f(\text{Image of a bird}; \theta) = 2$$

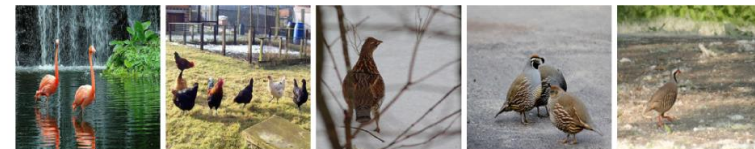
Supervised

Cat



...

Bird



...

Linear classifier

In this way we put a distance knowlaedge that we didn't meant, since the label at point 3 will seem similar to the one at point 4 than the one at point 10.

But numbers are simply numbers and they can chage as they are in a lookup table

$$f(x = \text{img} ; \theta) = 2$$

Bad idea! Classes are categorical variables, their assignment to integer ids is random and it is not designed to capture the semantic of classes (i.e. if cat is class 3 and we get class=2,5 out of f , it does not mean that x depicts half a bird and half a cat; or, similarly, the fact that two ids are nearby or far away does not mean that the corresponding classes share or not visual similarities)

$$f(\textcolor{red}{x}; \textcolor{green}{W}) = \textcolor{green}{W}\textcolor{red}{x} = \textcolor{blue}{class}$$

32x32x3=3072x1 CIFAR image

1x3072

Scalar

Linear classifier

$f(x =$  $;$ $\theta) = 2$

0 (plane)	45.4
1 (car)	128.3
2 (bird)	253
...	0.23
	-1.34
	4
	56
	-63
	78
	2

argmax → 2

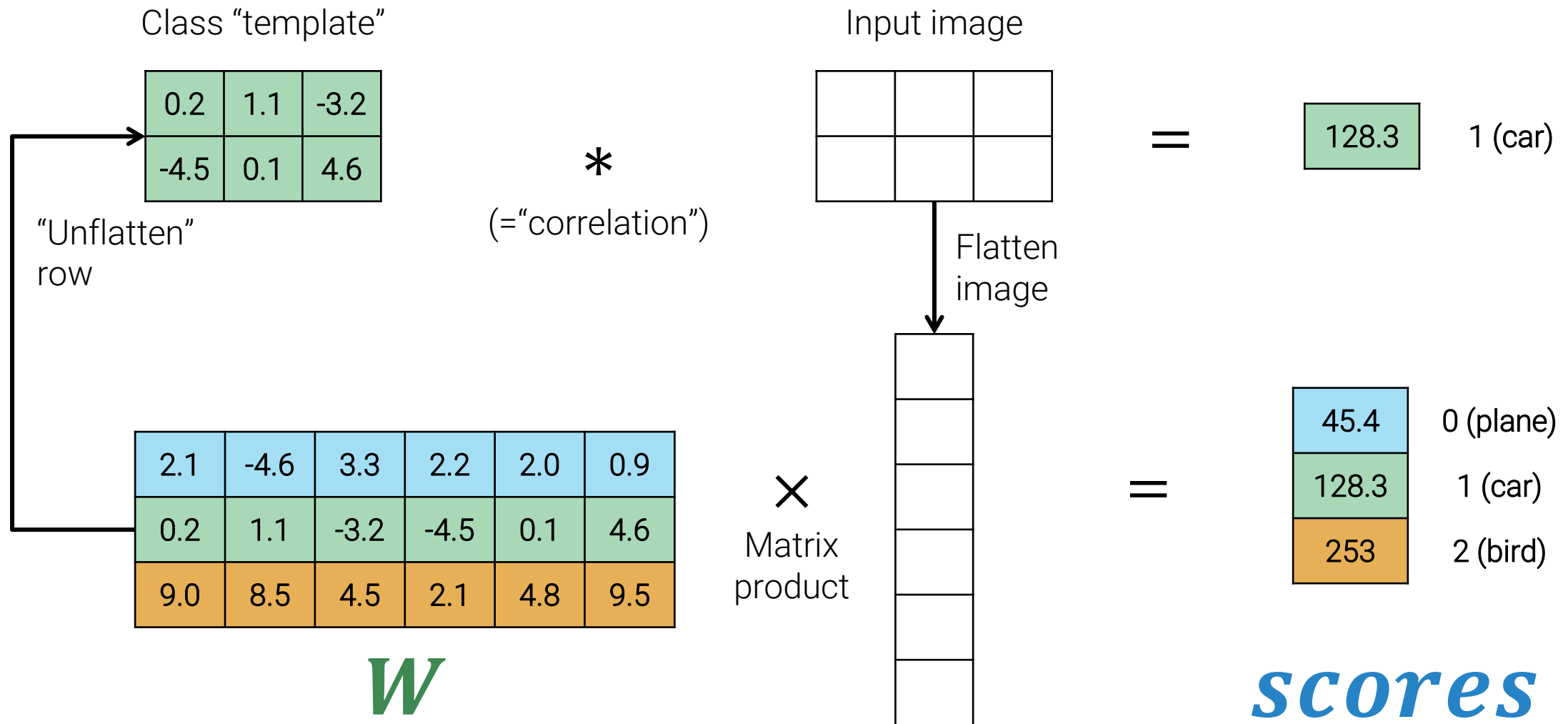
$f(\textcolor{red}{x}; \textcolor{green}{W}) = \textcolor{green}{W} \textcolor{red}{x} = \textcolor{blue}{scores}$
aka logits

32x32x3=3072x1 CIFAR image

10x3072
classes x pixels

10x1

Linear classifier as “template matching”



In ML, linear often means affine...

$$f(x = \text{img of bird} ; \theta) = 2$$

0 (plane)	45.4
1 (car)	128.3
2 (bird)	253
...	0.23
	-1.34
	4
	56
	-63
	78
	2

argmax

2

$$f(x; \theta) = \underset{\substack{\text{3072x1} \\ (W, b)}}{\overset{\parallel}{W}} x + \underset{\substack{\text{10x3072} \\ \text{Biases}}}{b} = \underset{\text{10x1}}{\text{scores}}$$

Learning as optimization

The space of the functions that a machine learning model can produce is its **hypothesis space** \mathbb{H}

Learning = solve an **optimization problem** to find the “best” **function** $h \in \mathbb{H}$

$$h^* = \operatorname{argmin}_{h \in \mathbb{H}} L(h, D^{train})$$

Parametric models

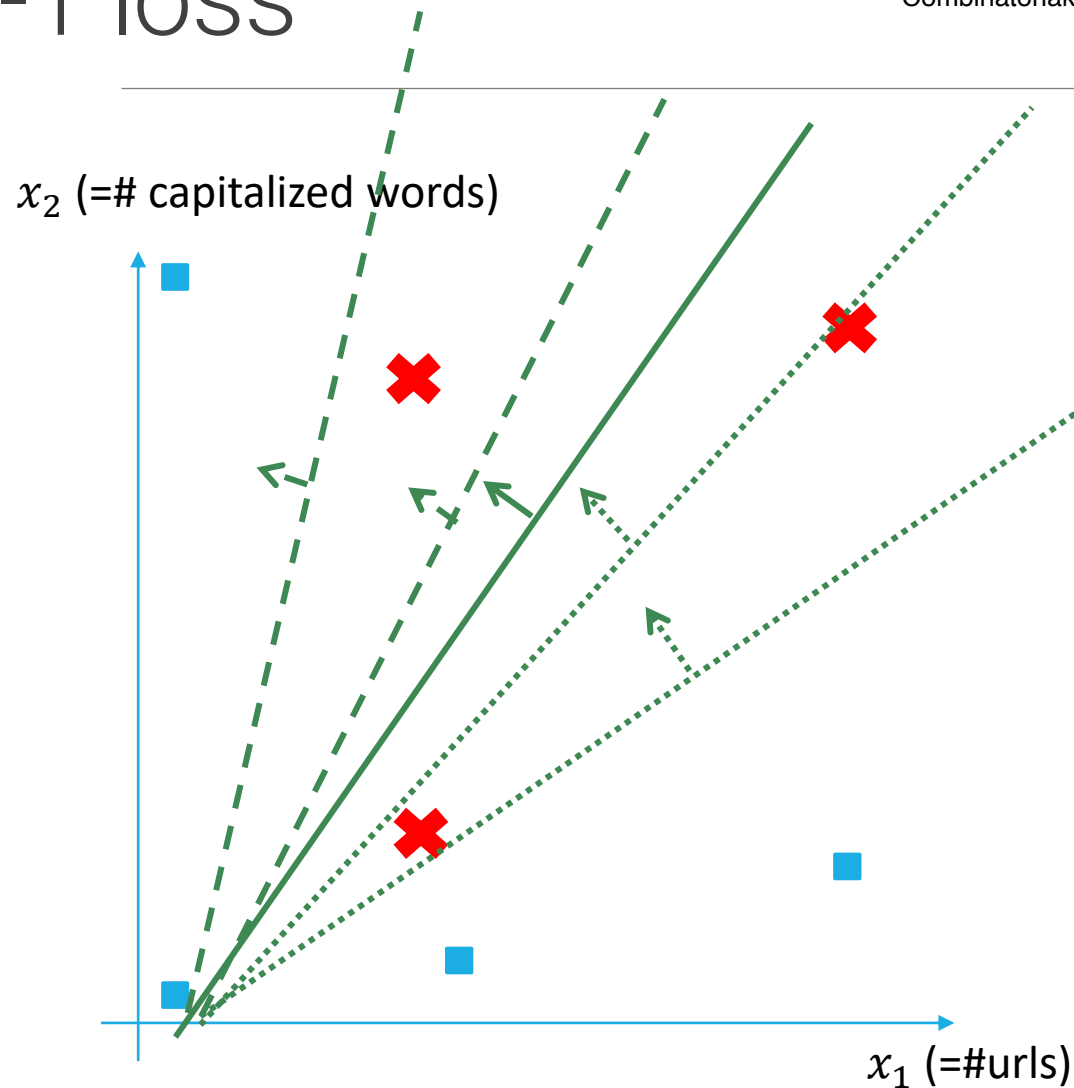
Learning = solve an **optimization problem** to find the “best” **parameters** $\theta \in \Theta$

$$\theta^* = \operatorname{argmin}_{\theta \in \Theta} L(\theta, D^{train})$$

What function L should we minimize?

0-1 loss

Combinatorial decision problem optimization, that scale poorly



$$L = \#errors?$$

This choice, known as the **0-1 loss**, results in a hard optimization problem.

Given the classifier represented by the solid line, changing it to become the dashed or the dotted line produces the same number of errors: **#errors is insensitive to small (and even large, sometimes) changes of the parameters.**

However, they are not equivalent directions of change: if we keep moving the classifier in the direction of the dotted line, we will improve it; whereas, if we keep moving in the direction of the dashed one, we will make more errors: **the error rate does not tell us if we are “moving” in the “right” direction while we change the parameters.**

We can minimize this loss as a combinatorial optimization problem, but then it does not scale well to large datasets

The loss function

Instead of directly optimizing accuracy, we then usually optimize a proxy measure, the **loss** function, that is **easier to optimize but still correlated with how good our classifier is**.

Loss function is also called **objective function, cost function, error function**,...

If the loss is high, our classifier is performing poorly, and we also expect low accuracy.

If the loss is low, our classifier is good, and we also expect high accuracy.

Hence, we prefer values of the parameters that minimize it on the training set

$$\boldsymbol{\theta}^* = \operatorname{argmin}_{\boldsymbol{\theta} \in \Theta} L(\boldsymbol{\theta}, \mathbf{D}^{train})$$

We will always work with losses whose value on a dataset is the average (or the sum) of the values for the single samples

$$L(\boldsymbol{\theta}, \mathbf{D}^{train}) = \frac{1}{N} \sum_i L(\boldsymbol{\theta}, (\mathbf{x}^{(i)}, \mathbf{y}^{(i)}))$$

What loss can we use for the linear classifier?

Historically, the RMSE between the prediction and the training label was used

$$L(\theta, (\mathbf{x}^{(i)}, \mathbf{y}^{(i)})) = L(\mathbf{W}\mathbf{x}^{(i)} + \mathbf{b}, \mathbf{y}^{(i)}) = \|\mathbf{W}\mathbf{x}^{(i)} + \mathbf{b} - \mathbb{I}(\mathbf{y}^{(i)})\|_2$$



0 (plane)	45.4
1 (car)	128
2 (bird)	253
...	0.23
	-1.34
	...

$$f(\mathbf{x}^{(i)}; \theta) = \mathbf{W}\mathbf{x}^{(i)} + \mathbf{b} =$$

0 (plane)	0
1 (car)	0
2 (bird)	1
...	0
	0
	...

$$\mathbb{I}(\mathbf{y}^{(i)}) =$$

This is called **one-hot encoding** of class bird

Softmax function

It turns out there are theoretical and practical reasons to prefer a loss that **transforms the scores computed by the classifier into probabilities** and then perform **maximum likelihood estimation** of θ .

How do we go from scores s_j to probabilities? We use the **softmax** function (softmax: $\mathbb{R}^n \rightarrow \mathbb{R}^n$)

$$p_{\text{model}}(Y = j | X = x^{(i)}; \theta) = \text{softmax}_j(s) = \frac{\exp(s_j)}{\sum_{k=1}^C \exp(s_k)}$$

0 (plane)	2,1	8,2	0,00
1 (car)	5,3	200,3	0,09
2 (bird)	7,6	1998,2	0,90
...	0,2	1,2	0,00
	-1,3	0,3	0,00
	2,5	12,2	0,01

Diagram illustrating the softmax function transformation. The input scores (left column) are transformed into unnormalized values (middle column) via the operation $\exp(s_k)$, and then normalized (right column) via the operation norm. .

Should be called “softargmax”, as it is a smooth and differentiable approximation of the one-hot encoding of the **argmax**

To implement it reducing numerical issues, it is useful to note that

$$\begin{aligned} \text{softmax}_j(s + c) &= \frac{\exp(s_j + c)}{\sum_k \exp(s_k + c)} = \\ \frac{\exp(s_j)\exp(c)}{\sum_k \exp(s_k)\exp(c)} &= \frac{\cancel{\exp(c)}\exp(s_j)}{\cancel{\exp(c)}\sum_k \exp(s_k)} = \text{softmax}_j(s) \end{aligned}$$

and then compute it as $\text{softmax}\left(s - \max_k s_k\right)$

Cross-entropy loss

Now that our linear classifier outputs «probabilities» over the classes, we can think of it as **a family of probability mass functions** over the classes given an image, indexed by the vector of parameters θ

$$\text{softmax}\left(f(\mathbf{x}^{(i)}; \theta)\right) \doteq p_{\text{model}}(Y | \mathbf{X} = \mathbf{x}^{(i)}; \theta)$$

Then, **the maximum likelihood estimation** of θ is

$$\begin{aligned}\theta^* &= \arg \max_{\theta} p_{\text{model}}(y^{(1)}, \dots, y^{(N)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}; \theta) \\ &= \arg \max_{\theta} \prod_{i=1}^N p_{\text{model}}(Y = y^{(i)} | \mathbf{X} = \mathbf{x}^{(i)}; \theta) \\ &= \arg \max_{\theta} \sum_{i=1}^N \log p_{\text{model}}(Y = y^{(i)} | \mathbf{X} = \mathbf{x}^{(i)}; \theta) \\ &= \arg \min_{\theta} \sum_{i=1}^N -\log p_{\text{model}}(Y = y^{(i)} | \mathbf{X} = \mathbf{x}^{(i)}; \theta)\end{aligned}$$

0 (plane)	0,00
1 (car)	0,09
2 (bird)	0,90
...	0,00
	0,00
	0,01

i.i.d. data

Avoid the «curse of products»

Per-sample cross-entropy loss
 $L(\theta, (\mathbf{x}^{(i)}, \mathbf{y}^{(i)}))$

If the true class of this image is “bird”, then the cross-entropy loss is $-\log(0.9) = 0.1$ (low value)
If it were “car”, then the loss value would have been $-\log(0.09) = 2.4$ (high value)

Putting everything together

Given

$$p_{model}(Y = j | X = x^{(i)}; \theta) = \text{softmax}_j(s) = \frac{\exp(s_j)}{\sum_{k=1}^C \exp(s_k)}$$

and the per-sample loss to minimize to perform maximum likelihood estimation

$$-\log p_{model}(Y = y^{(i)} | \mathbf{X} = \mathbf{x}^{(i)}; \theta)$$

the overall expression for the per-sample loss becomes

$$-\log \left(\frac{\exp(s_{y^{(i)}})}{\sum_{k=1}^C \exp(s_k)} \right) = -\log(\exp(s_{y^{(i)}})) + \log(\sum_{k=1}^C \exp(s_k)) = -s_{y^{(i)}} + \log(\sum_{k=1}^C \exp(s_k))$$

The second term of the sum on the right is usually referred to as the `logsumexp` and numerical libraries (e.g. PyTorch) usually have numerically stable functions to compute it. It approximates the max function, hence we can think of the cross-entropy loss as approximately

$$-\log \left(\frac{\exp(s_{y^{(i)}})}{\sum_{k=1}^C \exp(s_k)} \right) \approx -s_{y^{(i)}} + \max_k s_k$$

and of minimizing it as penalizing the most active incorrect prediction, i.e. a proxy to increase accuracy.

Losses in PyTorch

In PyTorch, you will find both `NLLLoss` (where NLL stands for negative log likelihood) and `CrossEntropyLoss`. Pay attention to the difference!

```
CLASS torch.nn.CrossEntropyLoss(weight: Optional[torch.Tensor] = None, size_average=None, ignore_index: int = -100, reduce=None, reduction: str = 'mean') [SOURCE]
```

This criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class.

It is useful when training a classification problem with C classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain raw, unnormalized scores for each class.

input has to be a *Tensor* of size either $(\text{minibatch}, C)$ or $(\text{minibatch}, C, d_1, \dots, d_K)$ with $K \geq 1$ for the K -dimensional case (described later).

```
CLASS torch.nn.NLLLoss(weight: Optional[torch.Tensor] = None, size_average=None, ignore_index: int = -100, reduce=None, reduction: str = 'mean') [SOURCE]
```

The negative log likelihood loss. It is useful to train a classification problem with C classes.

If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* given through a forward call is expected to contain log-probabilities of each class. *input* has to be a *Tensor* of size either $(\text{minibatch}, C)$ or $(\text{minibatch}, C, d_1, d_2, \dots, d_K)$ with $K \geq 1$ for the K -dimensional case (described later).

Obtaining log-probabilities in a neural network is easily achieved by adding a *LogSoftmax* layer in the last layer of your network. You may use *CrossEntropyLoss* instead, if you prefer not to add an extra layer.

The *target* that this loss expects should be a class index in the range $[0, C - 1]$ where $C = \text{number of classes}$; if

Where are we?



$$f(\underset{\substack{\uparrow \\ 3072 \times 1}}{x^{(i)}}; \underset{\substack{\parallel \\ (W, b)}}{\theta}) = \underset{\substack{\downarrow \\ 10 \times 3072}}{W} x + \underset{\substack{\downarrow \\ 10 \times 1}}{b} =$$

0 (plane)	2,1
1 (car)	5,3
2 (bird)	7,6
...	0,2
	-1,3
	2,5
	3,5
	4,5
	5,5
	6,5

$$s^{(i)} \rightarrow \boxed{-\log \left(\frac{\exp(s_{y^{(i)}})}{\sum_{k=1}^C \exp(s_k)} \right)} \leftarrow \mathbb{I}(y^{(i)})$$

0 (plane)	0
1 (car)	0
2 (bird)	1
...	0
	0
	0
	0
	0
	0
	0

Now we can express preferences over different choices of θ with one number. How can we use this knowledge to train our model, i.e. select the "best" θ ?

Gradient descent

$$\theta^* = \operatorname{argmin}_{\theta \in \Theta} \sum_i L(\theta, (x^{(i)}, y^{(i)}))$$

0. (Randomly) initialize $\theta^{(0)}$

for $e = 1, \dots, E$ epochs

1. **Forward pass:** classify all the training data to get the predictions $\hat{y}^{(i)} = f(x^{(i)}; \theta^{(e-1)})$ and the loss $L(\theta^{(e-1)}, D^{train})$
2. **Backward pass:** Compute the gradient $g = \frac{\partial L}{\partial \theta}(\theta^{(e-1)}, D^{train})$
3. **Step:** Update the parameters: $\theta^{(e)} = \theta^{(e-1)} - lr * g$

Important design decisions /
hyperparameters

Initialization method?

Number of epochs? Or
alternative way to decide to stop?

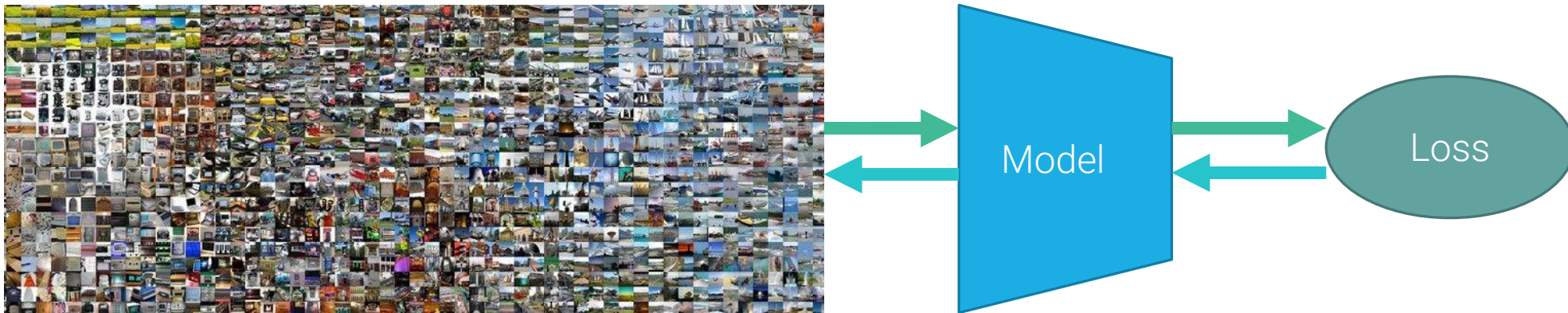
Learning rate lr ?

Limits of gradient descent

$$L(\theta, D^{train}) = \sum_i L(\theta, (x^{(i)}, y^{(i)})) \Rightarrow \nabla_{\theta} L(\theta, D^{train}) = \sum_i \nabla_{\theta} L(\theta, (x^{(i)}, y^{(i)}))$$

The total gradient $\nabla_{\theta} L$ is computed as the sum of the gradients of the single training samples.

Think about training a linear model on ImageNet (1.3M training images...).



1.3M forward passes and 1.3M backward passes just to perform one (usually tiny) step

Batch gradient descent is an approximation

ImageNet is large...
but all the possible images of its 1000 classes are a much larger set



The gradient computed by gradient descent approximates the “true” gradient, even when computed on all available training data, as they are a (large, but not full) sample of the true population (i.e., all the possible images of its classes)

SGD - Stochastic Gradient Descent

$$\theta^* = \operatorname{argmin}_{\theta \in \Theta} \sum_i L(\theta, (x^{(i)}, y^{(i)}))$$

A faster alternative to optimize a differentiable function is **stochastic gradient descent**.

0. (Randomly) initialize $\theta^{(0)}$

for $e = 0, \dots, E - 1$ epochs

1. Randomly shuffle examples in \mathbf{D}^{train}

for $i = 0, \dots, N - 1$

2. Forward pass: classify $\mathbf{x}^{(i)}$ to get the predictions $\hat{y}^{(i)} = f(\mathbf{x}^{(i)}; \theta^{(e*N+i)})$ and the loss $L(\theta^{(e*N+i)}, (\mathbf{x}^{(i)}, \mathbf{y}^{(i)}))$

3. Backward pass: Compute the gradient $g = \frac{\partial L}{\partial \theta}(\theta^{(e*N+i)}, (\mathbf{x}^{(i)}, \mathbf{y}^{(i)}))$

4. Step: Update the parameters: $\theta^{(e*N+i+1)} = \theta^{(e*N+i)} - \mathbf{lr} * g$

One sample instead of the full dataset
Sometimes this is also called "on-line" GD

Parameters updated after every forward+backward pass

SGD - Stochastic Gradient Descent with minibatches

A compromise: use a mini-batch of data of size B at each iteration instead of a single example. The number of parameter updates in each epoch will be $U = \left\lceil \frac{N}{B} \right\rceil$

0. (Randomly) initialize $\theta^{(0)}$

for $e = 0, \dots, E - 1$ epochs

1. Randomly shuffle examples in \mathbf{D}^{train}

for $u = 0, \dots, U - 1$

2. Forward pass: classify the examples $\mathbf{X}^{(u)} = \{\mathbf{x}^{(Bu)}, \dots, \mathbf{x}^{(B(u+1)-1)}\}$ to get the predictions $\hat{\mathbf{Y}}^{(u)} = \{\hat{y}^{(Bu)}, \dots, \hat{y}^{(B(u+1)-1)}\} = f(\mathbf{X}^{(u)}; \theta)^{(e*U+u)}$ and the loss $L(\theta^{(e*U+u)}, (\mathbf{X}^{(u)}, \mathbf{Y}^{(u)}))$

3. Backward pass: Compute the gradient $\mathbf{g} = \frac{\partial L}{\partial \theta}(\theta^{(e*U+u)}, (\mathbf{X}^{(u)}, \mathbf{Y}^{(u)}))$

4. Step: Update the parameters: $\theta^{(e*U+u+1)} = \theta^{(e*U+u)} - \mathbf{lr} * \mathbf{g}$

B samples are processed simultaneously
For $B = 1$ we recover online SGD,
for $B = N$ batch gradient descent

Design decisions /
hyperparameters

Initialization method?
Number of epochs? Or how to
decide to stop?
Learning rate, i.e. step size?
Mini-batch size?

Mini batches - tradeoffs

- Larger batches provide smoother estimations of the gradient, but with less than linear returns.
- Larger batches better exploit parallel hardware, below a minimum size no gain in training time.
- If examples in a batch are processed in parallel, memory requirements scale linearly with batch size.
- Usually **power of 2 sizes** (more suited for parallel hardware), typical values on single GPUs is 16 to 256, but modern distributed training can use values up to 8,192 or 16,384 on multiple GPUs. [Goyal et al. 2018]
- **Smaller batches may have a regularization effect** and results in better generalization, at the cost of longer training time. [Smith et al., 2020]

- Usual rule of thumb: **start with the largest power of 2 size that fits in the memory of your GPU**. Experiment from there, if time allows.

Goyal et al., Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, 2018. <https://arxiv.org/abs/1706.02677>
Smith et al., On the Generalization Benefit of Noise in Stochastic Gradient Descent, 2020. <https://arxiv.org/abs/2006.15081>

How to compute gradients?

Gradients can be computed

- Numerically

- Slow and approximate
- But easy to implement

- Analytically, by exploiting the rules of calculus and, in particular, the **chain rule**.

$$\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$$

- Exact, but slow, tedious, and error prone. For the linear classifier, we get

$$\frac{\partial L}{\partial b_j} = \frac{\partial}{\partial b_j} - \log \left(\frac{\exp(s_j)}{\sum_{k=1}^C \exp(s_k)} \right) = \frac{\partial}{\partial b_j} - \log \left(\frac{\exp(w_j x + b_j)}{\sum_{k=1}^C \exp(s_k)} \right) = - \frac{\sum_{k=1}^C \exp(s_k)}{\exp(w_j x + b_j)} \left(\frac{\partial}{\partial b_j} \frac{\exp(w_j x + b_j)}{\sum_{k=1}^C \exp(s_k)} \right) = \dots$$

- Automatically, by **automatic differentiation**, e.g. with the **backpropagation algorithm**

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986) "Learning representations by back-propagating errors." Nature, 323, 533–536.

What does a linear model learn on CIFAR10?

Accuracy about 38%

Let's use the template matching interpretation of a linear classifier to understand what the model is learning

It looks like **the background color is the predominant feature** used by the model

Moreover, **one template cannot capture multiple appearances** within one class, e.g. rotated cars, trucks, etc..

Distance between templates and images is a distance in input space.



What's next?

Learn more effective [representations](#).

